

AD-A265 869



RL-TR-93-8  
Final Technical Report  
March 1993

DTIC  
ELECTE  
JUN 16 1993  
S c D



# MULTICLUSTER

BBN Systems and Technologies

Edward F. Walker, Christopher E. Barber, James C. Berets,  
Susan K. Pawlowski, Jonathan Cole,  
and Natasha E. Cherniak

*APPROVED FOR PUBLIC RELEASE; DISTRIBUTION UNLIMITED.*

93-13403

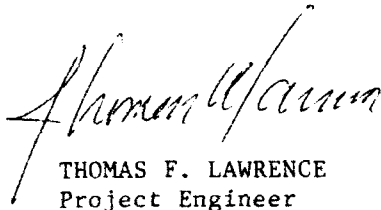


Rome Laboratory  
Air Force Materiel Command  
Griffiss Air Force Base, New York

This report has been reviewed by the Rome Laboratory Public Affairs Office (PA) and is releasable to the National Technical Information Service (NTIS). At NTIS it will be releasable to the general public, including foreign nations.


RL-TR-93-8 has been reviewed and is approved for publication.

APPROVED:



THOMAS F. LAWRENCE  
Project Engineer

FOR THE COMMANDER



JOHN A. GRANIERO  
Chief Scientist for C3

If your address has changed or if you wish to be removed from the Rome Laboratory mailing list, or if the addressee is no longer employed by your organization, please notify RL (C3AB ) Griffiss AFB NY 13441. This will assist us in maintaining a current mailing list.

Do not return copies of this report unless contractual obligations or notices on a specific document require that it be returned.

# REPORT DOCUMENTATION PAGE

Form Approved  
OMB No. 0704-0188

Public reporting burden for this collection of information is estimated to average 1 hour per response, including the time for reviewing instructions, searching existing data sources, gathering and maintaining the data needed, and completing and reviewing the collection of information. Send comments regarding this burden estimate or any other aspect of this collection of information, including suggestions for reducing the burden, to Washington Headquarters Services, Directorate for Information Operations and Reports, 1215 Jefferson Davis Highway, Suite 1204, Arlington, VA 22202-4302, and to the Office of Management and Budget, Paperwork Reduction Project (0704-0188), Washington, DC 20503.

1. AGENCY USE ONLY (Leave Blank)		2. REPORT DATE March 1993		3. REPORT TYPE AND DATES COVERED Final Sep 90 - Feb 92	
4. TITLE AND SUBTITLE MULTICLUSTER				5. FUNDING NUMBERS C - F30602-89-D-0056, PE - 63790D Task 3 PR - 6036 TA - QH WC - 06	
6. AUTHOR(S) Edward F. Walker, Christopher E. Barber, James C. Berets, Susan K. Pawlowski, Jonathan Cole, Natasha E. Cherniak					
7. PERFORMING ORGANIZATION NAME(S) AND ADDRESS(ES) BBN Systems and Technologies 10 Moulton Street Cambridge MA 02138				8. PERFORMING ORGANIZATION REPORT NUMBER 7725	
9. SPONSORING/MONITORING AGENCY NAME(S) AND ADDRESS(ES) Rome Laboratory (C3AB) 525 Brooks Rd Griffiss AFB NY 13441-4505				10. SPONSORING/MONITORING AGENCY REPORT NUMBER RL-TR-93-8	
11. SUPPLEMENTARY NOTES Rome Laboratory Project Engineer: Thomas F. Lawrence/C3AB/(315) 330-2805 Prime Contractor: TASC, 555 French Road, New Hartford, NY 13413-0895					
12a. DISTRIBUTION/AVAILABILITY STATEMENT Approved for public release; distribution unlimited.				12b. DISTRIBUTION CODE	
13. ABSTRACT (Maximum 200 words) In this report we present the implementation of multicluster operation in the Cronus distributed computing environment. A cluster is a collection of hosts on one or more networks. Clusters are formed to organize and control the relationship between organizations for reasons of security and administration. This document presents the mechanisms to support intercluster object addressing; these mechanisms form the basis for multicluster operation. Object-oriented distributed computing environments strive to present a uniform environment for building distributed applications by providing host and network transparent access to resources. Previously, the Cronus distributed environment operated in an environment of different types of hosts executing across several local area networks connected by gateways. Cronus achieved a uniform environment by treating the set of hosts as a single system in terms of authentication and in binding object identifiers to their locations. To extend Cronus to large environments, it is necessary to decompose the environment into collections of hosts. Decomposition is crucial to system scalability. It speeds up the location of objects and reduces the number of global name spaces.					
14. SUBJECT TERMS Cronus, Distributed Computing Environment, Cluster, Multicluster Access Control				15. NUMBER OF PAGES 36	
				16. PRICE CODE	
17. SECURITY CLASSIFICATION OF REPORT UNCLASSIFIED	18. SECURITY CLASSIFICATION OF THIS PAGE UNCLASSIFIED	19. SECURITY CLASSIFICATION OF ABSTRACT UNCLASSIFIED	20. LIMITATION OF ABSTRACT UL		

## MULTICLUSTER FINAL TECHNICAL REPORT

In this report we present the implementation of multicluster operation in the Cronus distributed computing environment. A cluster is a collection of hosts on one or more networks. Clusters are formed to organize and control the relationship between organizations for reasons of security and administration. This document presents the mechanisms to support intercluster object addressing; these mechanisms form the basis for multicluster operation.

Accession For	
NTIS CRA&I	<input checked="" type="checkbox"/>
DTIC TAB	<input checked="" type="checkbox"/>
Unannounced	<input type="checkbox"/>
Justification	
By	
Distribution /	
Availability Codes	
Dist	Avail and/or Special
A-1	

DTIC QUALITY INSPECTED

## Chapter 1. Introduction

Object-oriented distributed computing environments strive to present a uniform environment for building distributed applications by providing host and network transparent access to resources. Previously, the Cronus distributed environment operated in an environment of different types of hosts executing across several local area networks connected by gateways. Cronus achieved a uniform environment by treating the set of hosts as a single system in terms of authentication and in binding object identifiers to their locations.

To extend Cronus to large environments, it is necessary to decompose the environment into collections of hosts. Decomposition is crucial to system scalability. It speeds up the location of objects and reduces the number of global name spaces. Decomposition also accommodates the natural organization of computer installations in the real world.

This document first motivates multicluster Cronus by highlighting limitations of Cronus before clusters were supported. Then some basic definitions and design goals are presented. It then focuses on certain key design details, concentrating on the areas of communication and replication. We then present details of some of the new low-level communication protocols.

This document assumes the reader is acquainted with the Cronus distributed computing environment [BS89], [WFN90].

## Chapter 2. Limitations of Cronus Without Clusters

Without clusters, several difficulties presented themselves to independent organizations desiring to share services with each other. To accomplish sharing, organizations would have to "link up" their separate Cronus distributed environments into a single, inter-organizational environment. The problem was, once this was done, Cronus had no way to distinguish one organization's collection of hosts and services from another's. There was no straightforward way for one organization to specify that it did not want to share a particular service with another. If an organization wanted to share one of its services, it would have to allow access to all of its services. With clusters, on the other hand, an organization can readily prevent access to any of its services from other organizations.

Even more problematic was that Cronus only supported the notion of a single *configuration service* for a distributed environment. The configuration service plays an important role in Cronus; it serves as the depository and source of information concerning services, hosts, and --- most significantly --- what hosts run which services. Hosts use the information in the configuration service to determine the services they are to run. If organizations "linked up," they would have to decommission their existing separate configuration services, and manually merge their data and deposit it in a single new shared configuration service. But by doing so, each organization would give up control over its own assignment of services to hosts. If a user had permission to assign a service to his organization's hosts, there was no way to prevent him from assigning it to some other organization's host as well. By linking up, each organization gave up autonomy over its own configuration.

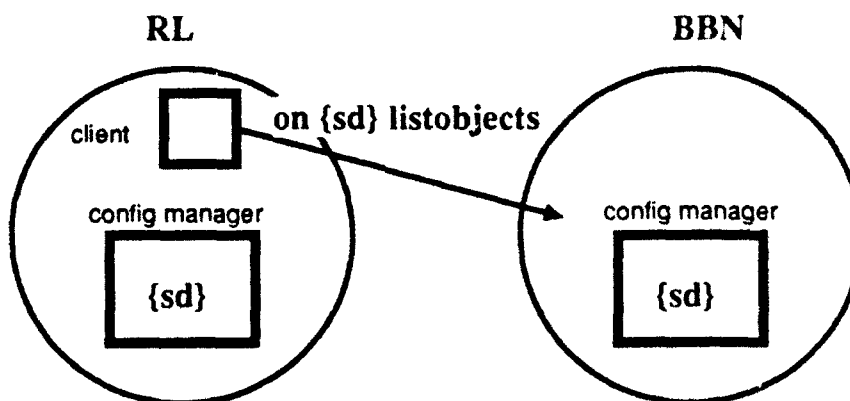


Figure 2-1: Anomalous Request Routing

If organizations linking up their environments chose to forego merging their separate configuration services into one, serious operational anomalies could occur. The basic problem was that the system could legitimately route requests intended for one configuration service to the "wrong" configuration service. A user trying to request a list of the services that run at his organization could instead get back a list relevant to some other organization. A host's request at boot time for the list of services it is supposed to run could go to the "wrong" configuration service.

The problem is that all these requests are invocations on a generic object. All configuration services manage the same generic objects. Figure 2-1 illustrates the problem. A user at one organization, *RL*, wants a list of services that run at his organization. To do this, he invokes the *ListObjects* operation on the generic service data object, *{sd}*. This operation will return the a list of all the service data objects stored at the replying configuration manager. (Each service data object contains information about one particular service.) *RL*'s configuration service manages the *{sd}* object, but so does the other organization's. Cronus can legitimately route the *ListObjects* request to *any* host with the *{sd}* object. As illustrated, the system chooses to route the request to BBN's configuration service. The user will get back a list of service data objects that describe BBN's services, not *RL*'s.

With clusters, Cronus supports the notion of several independent and separate configuration services within a single Cronus environment. Each organization can have its own configuration service, and therefore maintain complete autonomy over its own assignment of services to hosts. Invocations on generic objects can be constrained to be routed to those belonging to a particular organization.

Two organizations "linking up" would encounter problems with other services analogous to those with the configuration service. For example, the *authentication service*. Before clusters, only one authentication service was supported per distributed environment. An organization would either have to accept the operational anomalies that would occur if it did not give up its own authentication service, or give up its autonomous control over its principals and groups. Another problematic service would be the *directory service*, (formerly known as the *catalog service*). Again, each organization would have to merge its separate service into a single new shared service.

With clusters, Cronus circumvents all these problems. Each organization can have its own separate authentication service and directory service. By adding clusters to Cronus, the system can recognize and respect organizational boundaries.

## Chapter 3. Definitions, Goals, and Design Overview

Clusters allow boundaries to be erected between organizations. This chapter will define a *cluster* precisely. With that definition, we can examine some of the facilities that support inter-organizational cooperation and sharing.

### 3.1 Clusters

A *cluster* is a set of hosts grouped together into a single administrative unit. Each cluster is autonomous, responsible for its own administration and control. No host is permitted to be a member of more than one cluster.

Physical network layout does not have to play a part in determining host cluster membership. Two hosts on the same physical local area network can belong to different clusters. Two hosts widely dispersed and connected by a long-haul network can belong to the same cluster, as shown in Figure 3-1.

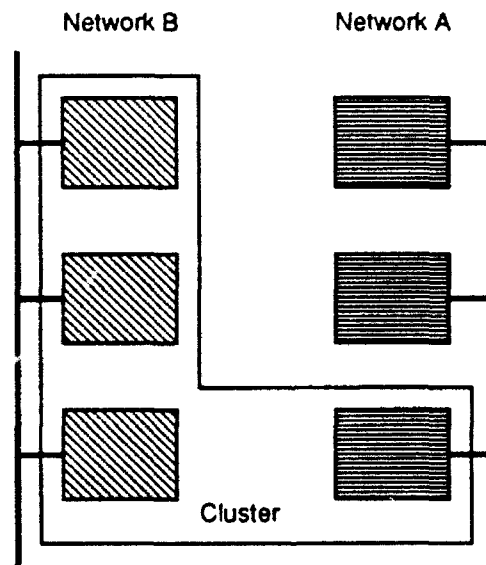


Figure 3-1: Network Topology and Cluster Boundary

Clusters are intended to allow administrative units to maintain control over their part of a Cronus environment, yet share services across cluster boundaries if so desired.



### 3.2 Sharing Services

A cluster is not an isolated unit: clusters can cooperate and share services with one another. If a cluster supports a service, it can permit other clusters to access that service. If a cluster does not permit access to a service, then operation requests from foreign clusters on objects managed by the service will be summarily rejected. If the cluster does permit access, the request is accepted as long as the requester has the necessary access rights as specified on the object's access control list.

Each cluster must explicitly enumerate which foreign clusters have access to its services. A service to which access by a foreign cluster is permitted is called an *exported service*, or simply an *export*. The service is said to be *exported* to the foreign cluster.

A cluster must explicitly declare that it wishes to access a service exported by a foreign cluster. An exported service to which a cluster desires access is called an *imported service*, or simply an *import*. The service is said to be *imported* from the foreign cluster.

If a client in cluster **B** wishes to access a service **S** in cluster **A**, then **A** must export the service to **B** and **B** must import the service from **A**. If the service is imported but not exported, the service will reject any access attempt from **B**. If the service is exported but not imported, hosts in cluster **B** will not be able to locate any of the objects. See Figure 3-2..

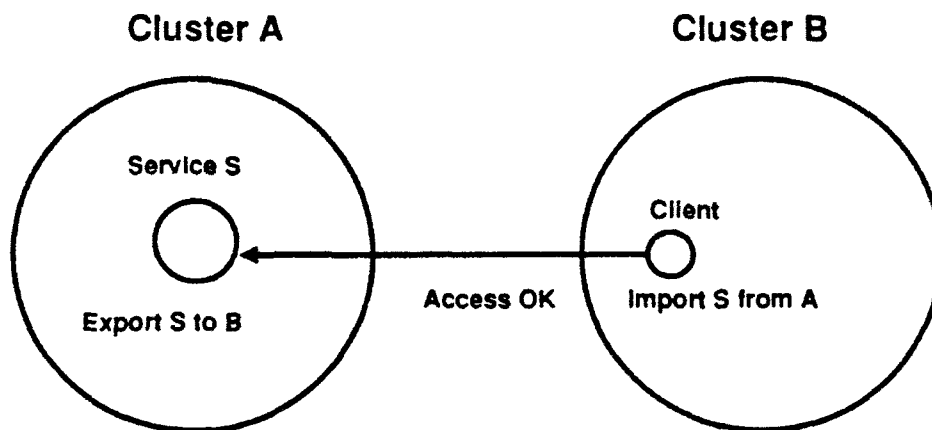


Figure 3-2: Importing and Exporting a Service

### 3.3 Replicating Services

A cluster can permit objects managed by its services to be replicated in foreign clusters. This is a stronger notion than merely permitting access to a service from foreign clusters. Permitting objects to be replicated across cluster boundaries implies that the service itself spans cluster boundaries, and thus several clusters must cooperate with regards to the administration of the service.

A cluster must explicitly declare the foreign clusters where objects of one of its services may be replicated. The foreign clusters are called the *domain* of the service.

For a replicated service to run successfully across cluster boundaries, all clusters involved in the replicated service must agree on the service's domain. If cluster **A** lists **B** as part of the domain of service **S**, then **B** must list **A** as part of the domain of **S**. Similarly, if **A** lists **B**, and **B** lists **C**, then **A** must also list **C**.

If a foreign cluster is a member of a service's domain, the service is implicitly considered to be exported and imported to the cluster.

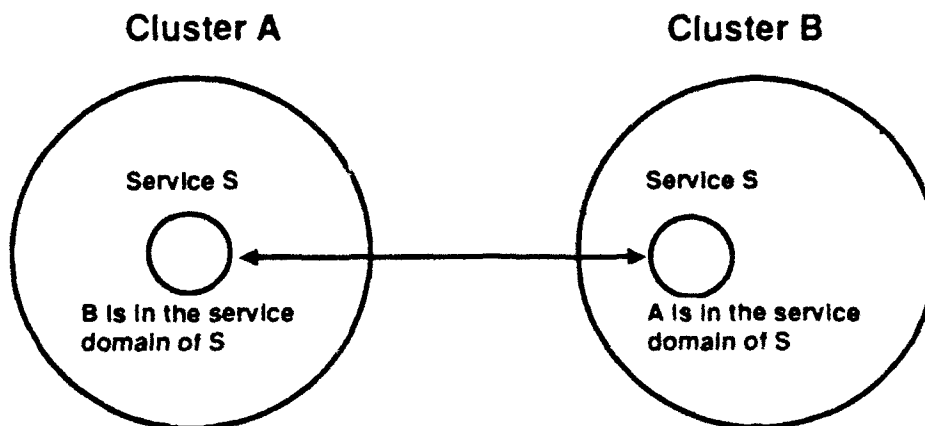


Figure 3-3: Service Domains

### 3.4 Configuration Control

In addition to providing configuration control within a cluster, the Cronus Configuration Manager has been extended to maintain a cluster's information about its imports, exports, and service domains.

#### 3.4.1 The Configuration Service

The information about a cluster's imports, exports, and domains is stored by the cluster's configuration service. Each cluster must run the configuration service, and it may not have any foreign clusters in its service domain. This restriction ensures that every cluster maintains autonomous control over its service configuration. The right to modify and create objects managed by the configuration service should be a closely guarded right, restricted to a few trusted and authorized individuals, and never granted to individuals whose "home" is a foreign cluster.

The Configuration Manager manages three types of objects: Host\_Data objects, Service\_Data objects, and Cluster\_Data objects.

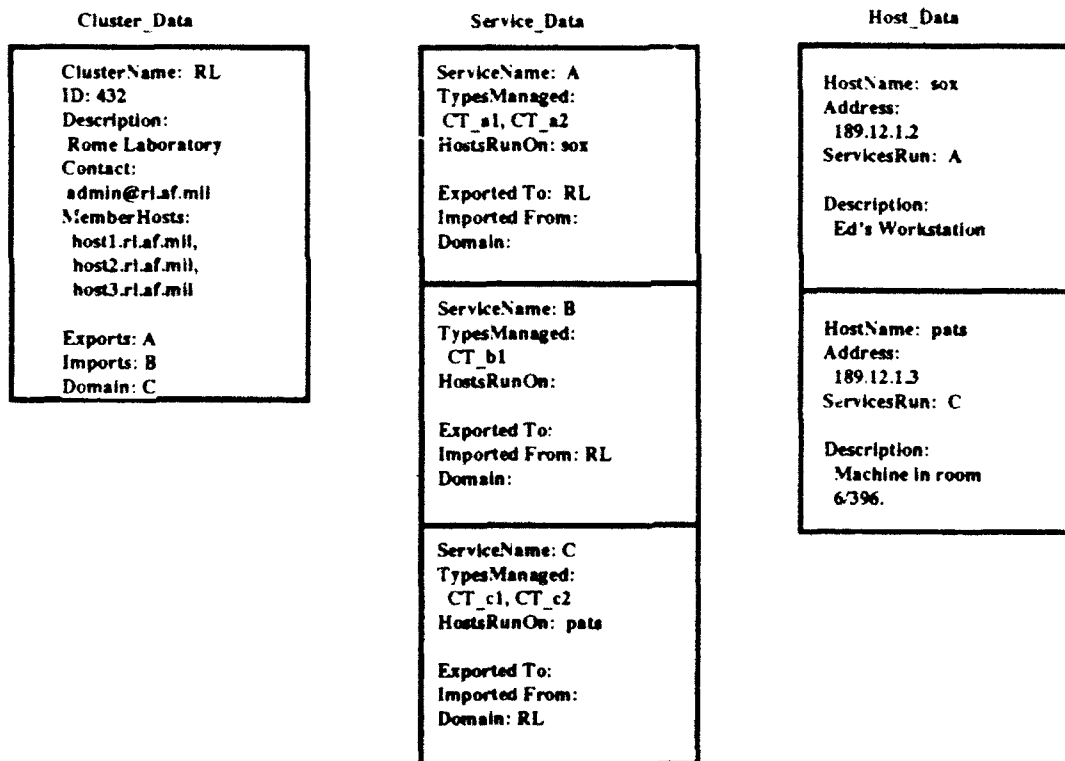


Figure 3-4: Example Configuration Database

A cluster's configuration service has a Host\_Data object for each host that is a member of the cluster.

The Host\_Data object for a given host contains the host's name, address, and a list of Service\_Data object identifiers identifying the services installed on the host.

A Service\_Data object records information about a service. It contains the service's name, the set of the types the service manages, and the set of Host\_Data objects identifying the hosts where the service is installed. It also contains three sets of Cluster\_Data objects. The first identifies the clusters the service is imported from, the second identifies the clusters the service is exported to, and the third identifies the clusters in the service's domain.

A Cluster\_Data object stores information about a foreign cluster. It contains the cluster's symbolic name and its unique cluster identifier. It also contains three sets of Service\_Data objects. The first identifies the services imported from the cluster, the second identifies the services exported to the cluster, and the third identifies the services whose domain includes the cluster. The Cluster\_Data object also contains a list of the IP addresses of some of the hosts belonging to the remote cluster. (This list provides a list of "contacts" in the remote cluster. As will be discussed, the multiclusterc location algorithm finds an object in a remote cluster by delegating the task to some host in the remote cluster. The host is chosen from the host list maintained in the Cluster\_Data object.)

### 3.4.2 Configuration Management Tool

Multiclusterc Cronus adds to the work required to administer a Cronus distributed computing environment. The administrator is no longer just responsible for maintaining his own organization's internal configuration of hosts and services. In Multiclusterc Cronus, the administrator must also maintain the service sharing relationships with outside organizations.

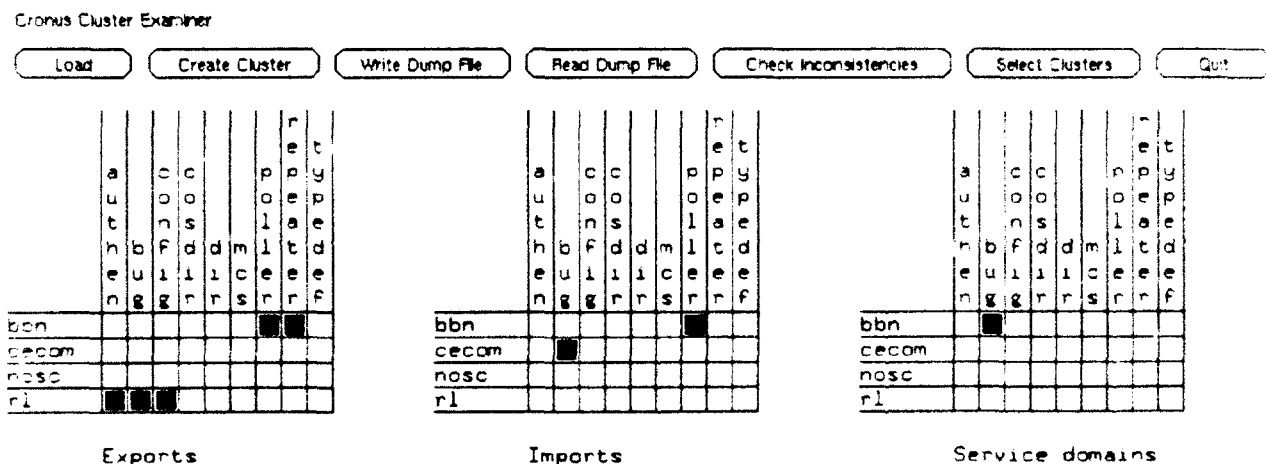


Figure 3-5: Configuration Management Tool

To lighten the burden of these added tasks on administrators, an interactive tool was developed. The main screen of the tool is shown in Figure 3-5. The names on the sides of the grids are cluster names; the names on the top are service names. A black square indicates the given service is exported (or etc)

to the given cluster. To export (or etc.) a service to a cluster, the administrator just clicks the mouse in the appropriate place in the grid. Similarly, to stop exporting (or etc.) a service to a cluster, the administrator just has to click on the appropriate blackened point in the grid. With the tool, it is very easy for an administrator to quickly view service sharing relationships with other clusters, and change those relationships if need be.

An additional complication caused by multicluster operation is the possibility of inconsistencies existing between clusters. The tool helps identify inconsistencies. Each cluster has completely autonomous control over its configuration. Although this is a desirable quality, it raises the very real possibility that clusters fail to cooperate properly and thus fail to establish consistent service sharing relationships between themselves. As has been discussed, it is important that service domains be consistent. For example, if cluster **A** lists **B** as part of the domain of service **S**, but **B** does not list **A**, an inconsistency exists. Inconsistencies in service domains can cause replicated services to malfunction. Another type of inconsistency involves imported services. A cluster can import a service from a foreign cluster that does not export it. If a service is imported, the administrator probably believes the service is accessible, and might like to be warned if this is not the case.

When the *Check Inconsistencies* button shown in Figure 3-5 is clicked, the tool checks for inconsistencies in service domains, and for imported services that are not exported. The tool does this by comparing the information in its own cluster's configuration service with that stored by each appropriate foreign cluster's configuration service. For the tool to be able to access a remote cluster's configuration service, it must be imported by the local cluster and exported by the foreign cluster. Once the tool is finished searching for inconsistencies, it displays any it finds to the user in the manner shown in Figure 3-6.

The screenshot shows a dialog box titled "Configuration Management Tool Error Report". At the top, it says "Number of errors found: 3". To the right of this text are two buttons: "Re-check" and "Cancel". Below the text is a section labeled "Fatal Errors" which contains a single message: "An inconsistency in the domain for service bug has been found in cluster bln". This message is displayed in a text area with a vertical scrollbar on the right. Below the "Fatal Errors" section is another section labeled "Errors" which contains two messages: "Cluster cecom imports service bug which is not exported by any cluster" and "Cluster bln imports service poller which is not exported by any cluster". This section also has a text area with a vertical scrollbar on the right.

Number of errors found: 3

Re-check Cancel

Fatal Errors

An inconsistency in the domain for service bug has been found in cluster bln

Errors

Cluster cecom imports service bug which is not exported by any cluster  
Cluster bln imports service poller which is not exported by any cluster

Figure 3-6: Configuration Management Tool Error Report

## Chapter 4. Key Design and Implementation Details

In this section we provide detailed discussion of several key algorithms and approaches that provide robust multicluster operation.

### 4.1 Object Location

Cronus supports location-independent invocation; the location of an object does not have to be supplied when invoking an operation on it. The system takes complete responsibility for finding a copy of the object and directing the invocation request to it. The part of the system that locates objects is called the *Locator*. The Locator is part of the Cronus Kernel. Once the Locator finds an object, information about its location is stored in the Kernel's *location cache*. The Locator is used only if relevant information is not first found in the location cache.

The Locator was completely redesigned and reimplemented for Multicluster Cronus. The old Locator was very simple, but suffered from some severe limitations. To find an object, the Locator would broadcast a *Locate* invocation on the object. The broadcast would be heard by every host on the network, (except those that dropped the incoming unreliable broadcast message). If a host had a copy of the object, it responded affirmatively. If a host did not have a copy, it did not respond. The Locator would wait for an affirmative response. If one was not received within about five seconds, the Locator reported a LOCATE\_FAILED error.

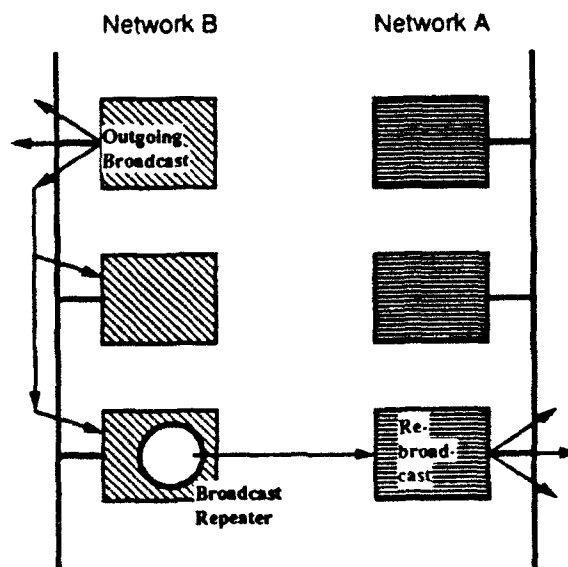


Figure 4-1: Broadcast Repeater

The old Locator relied on *Broadcast Repeaters* in Cronus environments encompassing more than one network. Network gateways do not forward broadcasts; a broadcast on one network is not heard on another. The old locator relied on every host hearing its broadcast *Locates*, regardless of the physical network topology. Cronus uses *Broadcast Repeaters* to make sure a broadcast on one network is heard on all other networks. A Broadcast Repeater is run on one host on each network. It is given a list of hosts, each on other networks, to repeat broadcasts to. When the Broadcast Repeater's host hears a broadcast, it is provided to the Broadcast Repeater. The Broadcast Repeater then transmits the message to all the hosts on its list. These hosts then rebroadcast the message. (They also tag the message so that it will not be repeated again.) See Figure 4-1.

The old Locator suffered from several problems and limitations. First, it was not robust. The old Locator relied on a single broadcast to find an object. In some environments it is not uncommon for the network to regularly drop broadcast messages. Second, the old Locator did not scale well, since it made every host process every locate. Third, it required the use of broadcast repeaters between networks, complicating the maintenance and administration of a Cronus environment. Fourth, the arbitrary five second wait for *Locate* replies was not appropriate for widely dispersed systems.

We established the following major requirements for the new multicluster Locator design.

1. The Locator must be robust, meaning that it must not fail to locate an available object. If an irrelevant host (not one where the requester or object resides) crashes, the location attempt should not fail.
2. The Locator must be efficient, meaning it must avoid communicating with hosts that do not have the object.
3. The Locator should find objects in a timely fashion.

The multicluster Locator satisfies these requirements. The old Locator, on the other hand, failed to satisfy the first two. In the situation where a Cronus environment encompassed more than one network, necessitating use of a broadcast repeater, a failure of the repeater host or the host repeated to could cause the old Locator to fail to find an available object. The old Locator certainly did not satisfy the second requirement, since it broadcast *Locate* requests to every host in the distributed environment.

The new multicluster Locator works basically as follows. The Locator is given the unique identifier (UID) of an object to find. The UID contains information identifying the type of the object. The Locator first extracts the type of the object from the UID. It then identifies the service that manages the type as follows. The Locator first checks a cache it maintains of service information. If this is unsuccessful, the Locator contacts the local cluster's configuration service. The configuration service keeps track of which services manage what types. The configuration service sends back information about the proper service, which the Locator then stores in its cache.

The information about a service that the configuration service returns to the Locator (and that is cached) includes the following: (1) a list of the hosts in the local cluster that run the service, (2) a list of the clusters the service is imported from, (3) a list of the clusters in the service's domain, and (4) a list of hosts in each cluster mentioned in items (2) or (3). This information comes from data stored in the configuration service's relevant *Service\_Data*, *Host\_Data*, and *Cluster\_Data* objects.

The Locator then makes use of the list of hosts in the (local) cluster that run the service. The locator sends each one of these hosts a *Locate* request, as illustrated in Figure 4-2. Here two hosts, *a.bbn* and *b.bbn*, are registered with the configuration service as running the service that manages the appropriate

type. (The *Locate* messages are transmitted using UDP, an unreliable protocol. The Locator times out and retransmits them to guard against message loss.)

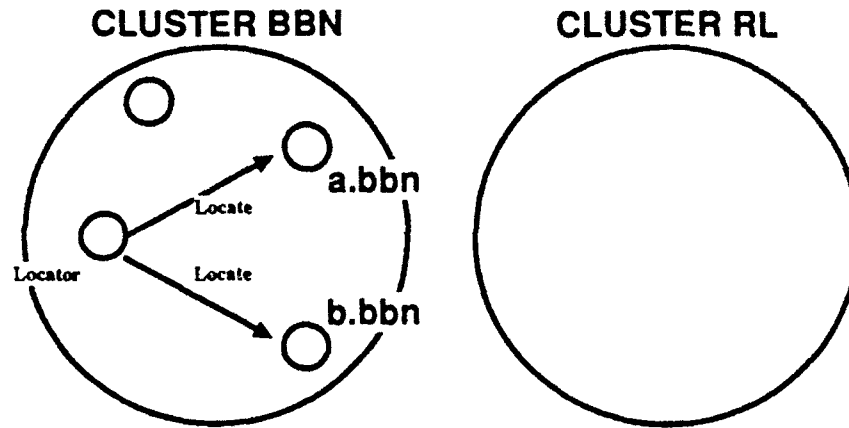


Figure 4-2: Contacting Local Cluster Hosts

The Locator then makes use of the rest of the information about the service to search for the object in remote clusters. A host in every cluster the service is imported from is sent a *ProxyLocate* request. Likewise, a host in every cluster in the service's domain is also sent a *ProxyLocate* request. The *ProxyLocate* operation is invoked on the generic Cronus\_Host object; the UID of the object being located is included as an argument. The Cronus Kernel manages the generic Cronus\_Host object; every Cronus Kernel manages such an object. (Every host runs the Cronus Kernel.) A Cronus Kernel that receives a *ProxyLocate* request looks for the specified object in its own cluster, and then sends back a reply indicating the object was found and its location, or else a negative reply. The *ProxyLocate* request is sent using a reliable protocol (TCP).

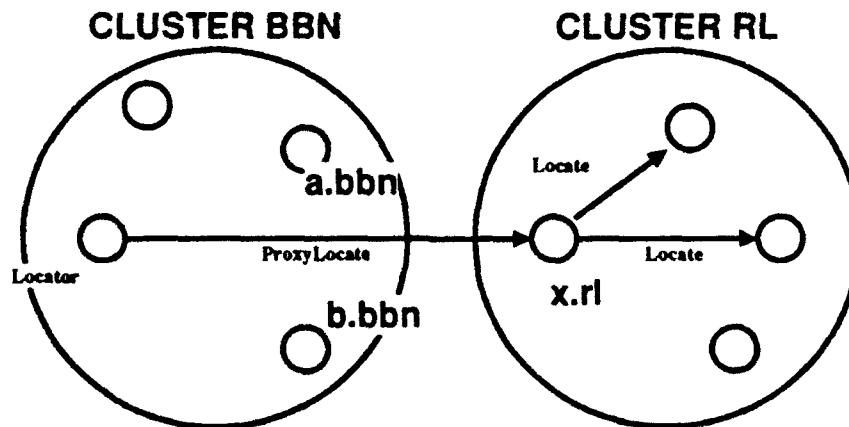


Figure 4-3: Contacting Remote Clusters



Figure 4-3 illustrates this step. In this case, the service is imported from cluster RL. The service information includes a list of some of the hosts in cluster RL, which includes the host "x.rl". The Locator chooses "x.rl" from the list and directs a *ProxyLocate* request to it. (The Locator's choice of a host from the list is fairly arbitrary.) "x.rl's" Cronus Kernel then has its Locator look for the object in just its own cluster. "x.rl's" Locator follows the same steps discussed above to accomplish what was illustrated in Figure 4-2. "x.rl's" Locator sends out *Locate* requests to each host in its cluster running the relevant service and collects the responses. If all are negative, it sends a negative reply back to BBN's Locator. If one is successful, a positive reply to the *ProxyLocate* is transmitted immediately.

If the attempt to send the *ProxyLocate* to "x.rl" fails because it is down, BBN's Locator selects another host from its list of hosts in cluster RL, and tries again.

BBN's Locator waits for responses to its *Locate* and *ProxyLocate* requests. The Locator stops with a successful outcome when the first positive reply is received. If all the replies are negative, the Locator becomes suspicious that its cached service information is out of date. If the service was installed on an additional host since the information was cached, or if the service was imported from an additional cluster, and if the object resides on the added host or in the added cluster, the Locator will obviously fail to find it.

The Locator checks if its cached service information is out of date by calling upon its local cluster's configuration service. The configuration service maintains an ascending *modification count* for each service. The Locator stores the *modification count* for each service in its cache. Checking if the cached information is out of date is a simple matter of comparing the actual versus the cached *modification count* for the relevant service. If the cache is out of date, it is updated, and the Locator tries to locate the object anew. If the cache proves to be up to date, the Locator reports back a *LOCATE\_FAILED* error.

A host processing a *ProxyLocate* also similarly checks if its cache is out of date if it fails to find the requested object.

A summary of the full Location algorithm follows. This summary includes some details not mentioned in the discussion above. The steps the Locator takes to find an object of type T are as follows:

1. The Locator broadcasts a "Locate" request for the object. The broadcast will only be heard by hosts on the local area network and any networks repeated to by a Cronus Broadcast Repeater.
2. The Locator starts listening for a positive response to any "Locate" (or "ProxyLocate") request. If one is received, the Locator immediately terminates this algorithm at that point, having successfully found a host that has the object.
3. The Locator looks in its location cache for information about T. The cache stores information previously obtained from the Configuration Manager about the location of the services that manage each type.
4. If nothing for type T is found in the location cache, the Locator continues with Step 8.
5. (Information for T was found in the location cache.) The cache information includes the list of the hosts running a manager of type T. If this list is empty, the Locator continues with Step 7. Otherwise, a "Locate" request is sent to each of these hosts, using the User Datagram Protocol (UDP). If a response is not received to one of these requests within a set time period, the request will be retransmitted.
6. The Locator waits a fraction of a second.

7. The cache information also includes the clusters in the service's domain and the clusters it is imported from. The cache also stores a list of hosts in each cluster. A "ProxyLocate" request is sent to a host in each cluster in the service's domain and to a host in each cluster the service is imported from using a reliable protocol. If the request fails because a host is unreachable, another host in the cluster is chosen and the request is retransmitted. The destination host processes the "ProxyLocate" by following this location algorithm, except that this Step and Step 1. are omitted, and reports back the result.
8. If the locate receives a positive response to a "Locate" or "ProxyLocate," then the object has been found. If not, then the Locator waits for a negative response to each "ProxyLocate," and for either a negative response to each "Locate" sent to a host within the local cluster or a timeout to occur.
9. Having failed to find the object, the Locator assumes that its cached location information is incorrect. It communicates with the Configuration Manager to see if its information for T is out of date. If not, the Locator terminates reporting LOCATE\_FAILED. If so, the Locator updates its cache and starts over with Step 3. If this step is reached again, the Locator immediately terminates and reports LOCATE\_FAILED.

The broadcast performed in Step 1 serves several purposes. First, the least loaded and/or "closest" manager with the object is likely to respond to the broadcast first, resulting in desirable load-balancing qualities. Second, before the Locator can obtain any location information from the configuration service, it must first locate a Configuration Manager. The broadcast provides a mechanism for finding a Configuration Manager. Third, the broadcast will locate objects that are not registered with the configuration service; this is a common case when new services are being developed and tested.

Several features of the location algorithm work to make it robust. In Step 5, a "Locate" request is sent to each host individually. This is an important robustness feature for clusters encompassing more than one local area network. If the broadcast "Locate" request was the only means of finding objects residing in the local cluster, the Locator would be susceptible to failures of the host running the Broadcast Repeater, or the host the Broadcast Repeater forwards broadcasts to. Neither of these hosts need be where the requester or object resides, and yet the failure of one of them could prevent the Locator from finding an object.

In addition, in Step 5 an individual "Locate" request is retransmitted if no response is received. Since the request is transmitted using an unreliable protocol, this guards against failing to find an object due to message loss.

The fashion in which the Locator searches for objects in foreign clusters also has several features that make it robust. In Step 7, the "ProxyLocate" request is transmitting using the normal Cronus reliable transmission protocol. This ensures that a reply (or an error) is eventually received. In addition, if the chosen host in the foreign cluster turns out to be down, the Locator then tries another. If the Locator did not do this, the crash of a host not containing the object could cause the object not to be found.

#### **4.1.1 Alternative Locator Design**

We had considered a Locator design that extended the Broadcast Repeater to a *Cluster Broadcast Repeater*. The basic idea was to simply repeat *Locate* broadcasts in remote clusters.

This scheme was attractive since it appeared to require relatively little implementation work; in particular the old Locator would not need modification.

There were some serious problems with this approach. First of all, this scheme violated our goal of efficiency for the Locator, since the Locator would require *every* host in *every* cluster to process a *Locate*. The multicluster Locator limits its search of the object to those clusters a service is imported from and to those in the service's domain. Secondly, the Cluster Broadcast Repeater would suffer from the same robustness problems as the original Broadcast Repeater: a crash of the host running the Repeater or of the host it repeated to would cause locates to fail. This violated our robustness goal for multicluster object location.

To try to fix the efficiency problem, we investigated adding a cluster ID to object UID's. The cluster ID would indicate the cluster in which the object resided. This scheme was rejected since it abandoned the previous stance in Cronus of not encoding an object's location in its UID. Placing a cluster ID in a UID would have made it difficult, if not impossible, to support migrating an object from one cluster to another.

## 4.2 Invocation Request Delivery Options

Cronus supports location independent invocation. A client does not have to supply any location information when invoking an operation on an object. The system itself finds the object and routes the invocation request to it.

In cases where more than one instance of an object exists, however, it is sometimes desirable to have control over which instances are eligible to receive the invocation. For example, when invoking an operation on the generic *Host\_Data* object, managed by every Configuration Manager, one might desire to have the operation processed by an instance in a particular remote cluster. A concrete example would be to obtain the list of hosts in a specific remote cluster.

The Cronus manager development tools generate RPC stubs that send operation requests to managers. The *INVOKECONTROL* argument in these stubs allows control over the instances eligible to receive a particular invocation. A *NULL* value, the common case, will result in the invocation being routed by the system to the first object instance it finds.

When an *INVOKECONTROL* structure is provided, three fields specify the instances eligible to receive the invocation: *HostUse*, *Host*, and *Cluster*. The interpretations of the *Host* and *Cluster* fields depends upon the value of the *HostUse* field. The semantics of the choices for the *HostUse* field are given below.

### HOST\_ANY

Any object instance can receive the invocation; the values of the *Host* and *Cluster* fields are ignored. This is the default behavior.

### HOST\_HINT

Same semantics as *HOST\_ANY*, except that the *Host* specifies a host where the sender believes the object to be.

### HOST\_DIRECT

Only the object instance at the host specified by the *Host* field is eligible to receive the invocation.

### HOST\_ANY\_IN\_CLUSTER

Any object instance residing in the cluster indicated by the *Cluster* field can receive the invocation; object instances in other clusters are ineligible.

#### **HOST\_HINT\_IN\_CLUSTER**

Same semantics as **HOST\_ANY\_IN\_CLUSTER**, except the *Host* field specifies the host where the sender believes the object resides.

#### **HOST\_ANY\_IN\_DOMAIN**

Any object instance residing in the service domain of the object's type is eligible to receive the invocation. Any object instance not residing in the local cluster's service domain for the type is ineligible to receive the invocation.

#### **HOST\_HINT\_IN\_DOMAIN**

Same semantics as **HOST\_ANY\_IN\_DOMAIN**, except the *Host* field specifies a host where the sender believes the object resides.

An example of an application that requires control over the instances eligible to receive an invocation request is the configuration management tool. One of the tool's functions is to check a foreign cluster's configuration information against the local cluster's for inconsistencies. In order to obtain a particular foreign cluster's configuration information, the tool must be able to specify that an invocation on a generic *Service\_Data* object is only to be sent to the a generic object in the foreign cluster. This ensures that a Configuration Manager residing in the foreign cluster answers the request.

**HOST\_ANY\_IN\_CLUSTER** is implemented as follows. Each location cache entry consists of an object's UID, the address of a host it resides on, and the id of the cluster the host belongs to. When a request is sent using **HOST\_ANY\_IN\_CLUSTER**, the kernel first tries to use the location cache to determine the target host. The kernel looks for an entry in the location cache that matches both the object's UID and the proper cluster id. If one is not found the Locator is asked to find the object, with the condition that the discovered location belong to the proper cluster. The Locator basically follows the steps it normally does, except it omits any steps that send messages to clusters other than the one specified.

**HOST\_ANY\_IN\_DOMAIN** is supported in a similar fashion, except here there are several clusters the target host is permitted to belong to (in general) instead of just one. The Cronus Kernel determines the clusters in the service domain by consulting its cached information for the service that manages the type of the relevant object. (The local cluster is always considered implicitly included in the service domain.)

**HOST\_HINT\_IN\_DOMAIN** (and **HOST\_HINT\_IN\_CLUSTER**) are implemented by first sending the request to the supplied hinted host address. If the hint turns out to be incorrect, the message is returned to the original sender. The original sender then changes the *HostUse* to **HOST\_ANY\_IN\_DOMAIN** (or **HOST\_ANY\_IN\_CLUSTER**) and starts over.

In addition to directing an invocation to a single object instance, one might want to broadcast a message to several instances of an object. The following *HostUse* field values specify that an invocation request is to be distributed to several object instances and gives control over the set of instances eligible to receive the request:

#### **HOST\_ALL**

A copy of the invocation is sent to all instances of the object.

#### **HOST\_ALL\_IN\_CLUSTER**

A copy of the invocation request is sent to all instances of the object in the cluster specified by the value of the *Cluster* field.

### HOST\_ALL\_IN\_DOMAIN

A copy of the invocation request is sent to all instances residing in the local cluster's service domain for the object type.

Reliable delivery is not provided when a request is broadcast to several instances. The request might be received by some eligible instances, but not by others.

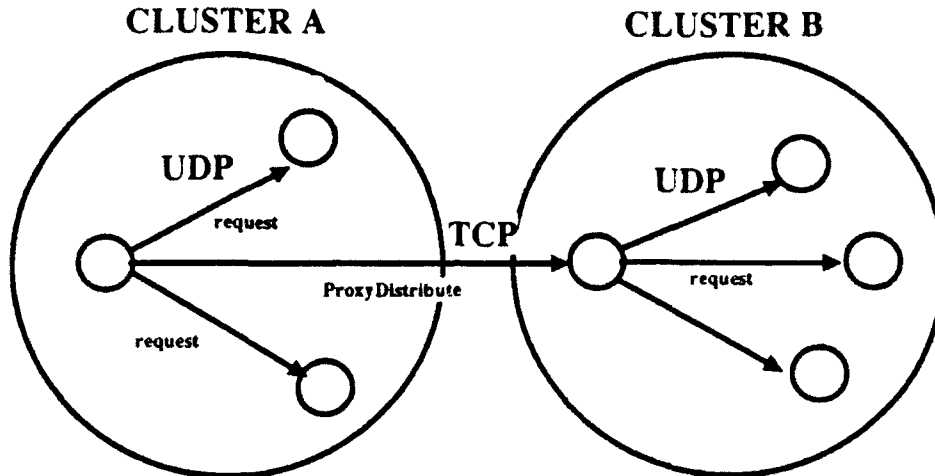


Figure 4-4: HOST\_ALL Implementation

HOST\_ALL is implemented as follows. The kernel looks up the service for the type of the target object in its service information cache. If there is no information for the type in the cache, the request is broadcast.

If there is information for the type in the cache, the HOST\_ALL request is sent as follows. The cached service information includes the list of hosts in the local cluster that run the service. Each one of these hosts is sent the message via UDP, an unreliable protocol. The cached service information also includes the list of clusters the service is imported from, plus the clusters in the service's domain. A host in each one of these clusters is sent a *ProxyDistribute* request via TCP, a reliable protocol. The *ProxyDistribute* is an operation invocation on the generic Cronus\_Host object; it includes the HOST\_ALL request as an argument. The generic Cronus\_Host object is managed by the Cronus Kernel. A kernel that receives a *ProxyDistribute* request uses its cached service information to send the HOST\_ALL request to each host in its cluster via UDP. If the kernel has no cached information for the type, it broadcasts the HOST\_ALL request. See Figure 4-4.

If the cached service information is out of date, the message might not be sent to all the managers it should be. This is acceptable, since HOST\_ALL is specified to be unreliable; delivery is not guaranteed. To prevent a host's cache from being out of date for a long period of time, the configuration service notifies each host in its cluster individually whenever a service's configuration is altered.

HOST\_ALL\_IN\_CLUSTER and HOST\_ALL\_IN\_DOMAIN are implemented in a fashion similar to HOST\_ALL. The same steps are used as in the HOST\_ALL case, except messages going to clusters other than the desired ones are not actually transmitted.

### 4.3 Intercluster Replication

The Cronus replication algorithm required only minor enhancements to be able to accommodate services that are replicated across cluster boundaries.

When a replicated manager is first installed, it needs to contact one of its peers in order to obtain local copies of replicated objects. To find a peer, the old algorithm broadcasted a locate request. In multicluster Cronus, this approach is inadequate since broadcasts are not heard across cluster boundaries. Instead, the locate is "logically broadcasted" throughout the service domain, by using the new `HOST_ALL_IN_DOMAIN` request delivery option.

A peer replicated manager needs to be able to manipulate its fellow peer managers. For example, it must be able to transmit updated objects to its peers in order to keep their copies current. In order to accomplish this, each peer needs access control permission to carry out the necessary operations on its peers. This access was previously achieved by requiring each peer run under the same Cronus principal UID. (Managers, like all processes running under Cronus, run under some Cronus principal.) The Cronus access control algorithm permits a caller running under the same principal as the callee to perform any operation.

We chose not to require that each peer run under the same principal in multicluster Cronus. We felt it was desirable to support the case where each peer's principal was managed by that peer's cluster's authentication service. Cronus principal objects cannot be replicated across cluster boundaries. To ensure cluster autonomy, multicluster Cronus restricts the service domain of the authentication service to a single cluster. (This is also true of the configuration service.) Therefore if peers in different clusters are to run under locally managed principals, peers must be permitted to run under different principals.

To permit this, for a service whose domain includes remote clusters, the configuration service stores a list of principals used by peers residing in remote clusters. A manager obtains this list of peer principals, and if a caller's principal is on the list, the caller is permitted to invoke any operation.

### 4.4 Kernel Virtual Reliable Connections

Unlike older implementations of Cronus, multicluster Cronus does not place any limit upon the number of reliable connections that may be open at one time. Multicluster Cronus accomplishes this by multiplexing a limited number of physical TCP connections among reliable *virtual* connections. This is done in a fashion that protects against undetected message loss and message reordering on the virtual connection.

Most of the operating systems Cronus runs on impose a limit on the maximum number of TCP connections a process can have open at a given moment. In particular, most versions of Unix allow no more than about sixty concurrently open connections. In older implementations of Cronus, once this limit was reached, attempts to establish new connections resulted in "host inaccessible" errors. Normally this error is reported only when the destination host is down or the network has failed.

In multicluster Cronus, it was suspected that the connection limit could be especially problematic, because of the connections used by the Locator. The multicluster Locator makes use of reliable connections to communicate with remote clusters. Therefore a kernel could need several connections to talk to remote cluster hosts, plus several more that remote clusters hosts could have established to it to use it as a proxy for locates. Therefore a Cronus kernel potentially might have to have a substantial number of connections open just to support locates, leaving few connections left for actually transmitting

application requests to their ultimate destinations. In any case, the restriction on open connections is unpalatable since it effectively restricts the number of remote hosts with which an application can reliably communicate.

Each virtual connection has its own message queue. When a virtual connection has no assigned TCP connection, any messages that to be transmitted on the virtual connection are merely added to the end of the queue. When a virtual connection does have a TCP connection assigned to it, messages to be transmitted are still added to the end of the message queue, but when the TCP connection is ready to accept data for transmission, the first message is taken off the queue and transmitted.

Whenever there is a virtual connection without a TCP connection, the system chooses a virtual connection with an assigned TCP connection and starts the process of deassigning it. Once the TCP connection is deassigned and closed, the waiting virtual connection is allowed to open a TCP connection and start transmitting its queued messages. A TCP connection is deassigned by placing a marker at the end of the message queue. Once the messages before the marker in the queue are all transmitted, the connection is ready to be closed. At this point, no more queued messages will be transmitted using the TCP connection.

The connection closing algorithm uses a handshake to ensure that no messages are lost. First a "Ready to Close" message is sent to the destination host on the other end of the connection. Once the other end receives this message, it starts the process of deassigning the connection if it has not already done so. Eventually the other end sends back a "Ready to Close" message. At this point, an "OK to Close" message is transmitted. An "OK to Close" message is transmitted whenever one side of a connection has itself transmitted a "Ready to Close" and has received an "Ready to Close" message from the other end of the connection, signifying that both ends of the connection have agreed not to use the connection further. (Thus the other side transmits an "OK to Close" message at this point as well.) The first side that receives an "OK to Close" message closes the connection. After transmitting the "OK to Close" message, an "OK to Close" message is either received or the connection is broken, the latter if the other end processes the "OK to Close" before this end gets its "OK to Close." When the connection is closed or broken, it frees up a descriptor so that a TCP connection can be established for a virtual connection that needs one.

The algorithm uses a queue marker when deassigning a physical connection to ensure that progress is made with regards to message transmission. Using the marker avoids a possible situation where the system could thrash constantly deassigning and assigning TCP connections to virtual connections, but without any actual application messages being transmitted. The marker ensures that all messages that have been waiting for a virtual connection to get a real connection get transmitted once the virtual connection is awarded a real connection.

When opening a physical connection, it may be the case that the target host does not have an available TCP connection. Every host tries to keep a TCP connection free at all times. When a connection is made that causes the last connection to be used, a "refusal" message is immediately transmitted down the connection and it is then closed. When a host sends such a "refusal" message, it opens a virtual connection to the source host, so that eventually the refusing host will itself initiate a physical connection to the source host. A host that is refused a connection earmarks an available connection only to be used when the refusing target host eventually initiates a connection. This ensures that the connection will be successfully set up when the intended target initiates the connection. This ensures that two hosts do not thrash constantly trying to connect to the other and being refused each time.

Setting an available connection aside when a connection is refused creates the potential for deadlock. The worst situation that could occur would be all hosts setting aside all available connections. To avoid deadlock, a certain number of physical connections are set aside only to be used to accept incoming connections and make outgoing connections to formerly refused hosts.

## 4.5 Directory Sharing

The *directory service* supports a hierarchical symbolic name space for Cronus objects. (The directory service has in the past also been known as the *catalog service*.) See Figure 4-5. The leaves of the name space tree are entries containing arbitrary object UID's. Pathnames are formed by separating the component names with ":'s." For example, in Figure 4-5, the path name `:cronus:config:hosts:pats` resolves to the identifier of a particular Host\_Data object, presumably the one that contains information about the host named *pats*.

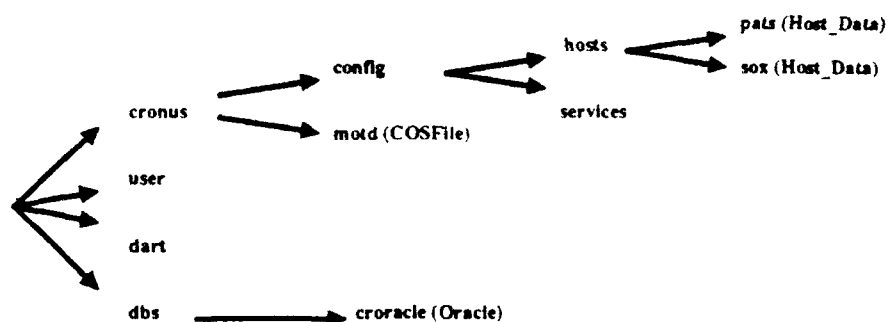


Figure 4-5: Directory Name Space

To ensure that a cluster has autonomous control over its own directory name space, the service domain of the directory service is limited to a single cluster. The *same* directory cannot be stored in two different clusters.

In multicluster Cronus, one can *mount* a remote cluster's directory name space. This makes the remote cluster's entire name space appear to be hung off an arbitrary path name in the local cluster's name space. In Figure 4-6, cluster BBN's entire name space is hung off cluster RL's path name `:rmt:bbn`. For example, a RL user can use the pathname `:rmt:bbn:dbs:croracle` to look up the object stored in BBN's name space under `:dbs:croracle`. RL's pathname `:rmt:bbn` is called a *mountpoint*.

Mountpoints are easily created by using the *createmount* command. The mountpoint shown in Figure 4-6 was created by the administrator typing *createmount bbn :rmt*. For a remote directory to be mounted successfully, the mounting cluster must import the directory service from the remote cluster, and the remote cluster must export it to the mounting cluster.

Mountpoints simplify accessing remote cluster directory name spaces. Without mountpoints, users would have to explicitly specify the cluster whose directory service should resolve a given path name. When mountpoints are used, path names can always be submitted to the local cluster's directory service. Mountpoints also permit symbolic links from the local cluster's name space into the remote cluster's name space to be readily supported.



Mountpoints are implemented as follows. Mountpoint entries in the directory name space contain the UID of the root directory in the remote cluster, and an indication that it is a mountpoint, rather than a normal directory. When the directory service encounters a mountpoint when resolving a pathname, it invokes an operation on the foreign root directory to resolve the rest of the pathname.

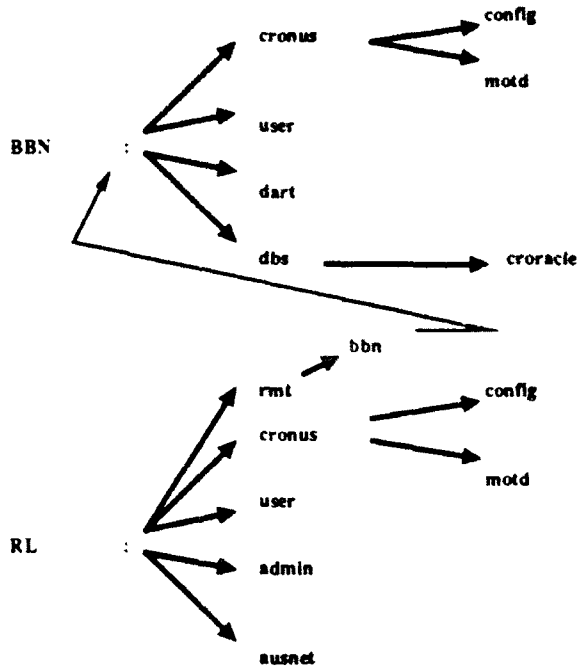


Figure 4-6: Mounting a Remote Directory

## 4.6 Cronus Commands

Most of the Cronus commands had to be upgraded to function properly in a multicluster environment. The command implementations were changed to make sure requests were processed in the proper cluster. For example, the *displayservice* command displays a list of all services. The desirable behavior for *displayservice* is that it lists the services known to the local cluster's configuration service. This command is implemented by invoking the *ListObjects* operation on the generic *Service\_Data* object. As was illustrated in Figure 2-1, by default a request on a generic object can legitimately be routed to any cluster. The *displayservice* command's implementation must be careful to constrain the generic *Service\_Data* objects eligible to receive the *ListObjects* request to those residing in the local cluster. Therefore, the *ListObjects* on the generic *Service\_Data* object was changed to use the *HOST\_ANY\_IN\_CLUSTER* delivery option discussed in section 4.2.

A */cluster* qualifier was also added to several commands. This qualifier was added to commands that normally display information concerning the local cluster. The qualifier permits the user to instead force the command to display information about any desired cluster. For example, the *displayservice* command supports the */cluster* qualifier. The command *displayservice /cluster=bbn* lists all the services defined by cluster bbn's configuration service.

## 4.7 Authentication

Multicluster Cronus provides privacy: the protection against unauthorized release of data. Data in messages can be protected by encrypting it with a secret key known only to the sender and receiver. Assuming that the encryption algorithm is secure and that keys are not divulged, no unauthorized party can decrypt a message.

Multicluster Cronus also provides for authentication: the assurance that when two parties interact with one another, each is certain of the other's identity. Only hosts that have been registered with the authentication service can successfully issue and process secure requests.

Just as each cluster has autonomy with regards to its configuration (e.g., the assignment of services to hosts), each cluster also has autonomy with regards to authentication. Each cluster has the responsibility of creating and maintaining principals for its members, and for safeguarding their secret keys. Each cluster must run the authentication service. The authentication service may not have any foreign clusters in its domain. This ensures that each cluster maintains sole control over its authentication information.

### 4.7.1 Request Security

There are three types of security for requests and their associated replies, listed in order of increasing safety. The *Security* field of the INVOKECONTROL structure is used to specify a request's security. The field may contain any of the following values:

#### SEC\_NONE

Self-explanatory.

#### SEC\_AUTHENTICATED

Messages contain an authenticator.

#### SEC\_PRIVATE

Messages contain an authenticator and the data portion will be encrypted.

### 4.7.2 Ticket and authenticator formats

The format of the *tickets* and *authenticators* used in the authentication scheme are as follows:

Kernel to Kernel Ticket - encrypted by target host's secret key

HOSTNUM	Sending Host
CLUSTERID	Sending Cluster
DATE	Expiration Time
U32I	Random number
OVEC	Session key

Kernel to Kernel Authenticator - encrypted by session key

HOSTNUM	Sending Host
CLUSTERID	Sending Cluster
DATE	Timestamp
U32I	Sequence number

#### 4.7.3 Secret keys

The following types of keys are maintained:

##### **KERNEL KEY**

Unique to a particular kernel and known to that kernel and its cluster's authentication service.

##### **SESSION KEY**

Unique to a particular link between two specific kernels. Session keys will be used in both directions. These keys will be generated by an Authentication Manager and handed to the kernels in a Kernel to Kernel ticket, described in section 4.7.5.3.

##### **INTER-CLUSTER AUTHENTICATION KEY**

Unique to a particular pair of clusters. Known to the Authentication Managers of both clusters and used for private communications between them.

#### 4.7.4 Special Groups

A special authentication group is supported to simplify cluster administration, the *CronusOperators* group. The UID of this group is a well-known constant. This group existed in Cronus before cluster support was added. The CronusOperators group essentially defines a Cronus environment's *super* or *privileged* users. Multiclustercronus prevents one organization's super users from also automatically being considered super users in other organizations.

Each cluster is responsible for designating certain principals as being CronusOperators. This is done by using the conventional facilities of the authentication service for establishing group memberships. By convention, CronusOperators are allowed to modify authentication and configuration service information as they desire. (The Cronus installation scripts automatically set this up.)

Multiclustercronus prevents one cluster's CronusOperators from being able to modify administrative data in another cluster. If a request comes from a remote cluster, and its sender claims to be a member of CronusOperators, this claim is ignored. The mention of the CronusOperators group in the sending process's access group set (AGS) is deleted by the receiver before performing access checks.

#### 4.7.5 Protocols

The following sections provide detailed outlines for several of the low-level communications protocols used in multiclustercronus. The notation "[packet]somekey" is used to mean that the packet (or whatever is in the brackets) is encrypted with "somekey" as the encryption key.

##### **4.7.5.1 Outgoing Reliable Request**

1. Client library sets *Security* field in the Cronus request header from the value specified in the INVOKECONTROL structure.
2. The library sends the message in cleartext to the kernel:

<Cronus Header><Request Header><Data Header><Data>

3. The kernel determines the target host for the request, by either consulting the location cache or consulting the Locator.
4. If the target host is the current host then no further security operations will be performed; the request is handed to the proper manager.
5. The message is then sent to the target host using the procedure in 4.7.5.2.

#### 4.7.5.2 Outgoing Reliable Message

This procedure is used to send both request and reply messages reliably to remote hosts. If the "remote host" is the same as the sending host, the following procedure is dispensed with and the message is immediately handed to the appropriate local process. The security of a reply message is set to match that of its matching request.

1. If a reliable connection has not already been established to the target host, then one is opened. This connection is initially not secure.
2. If the message is SEC\_NONE the kernel immediately sends the message down the connection. If the connection was previously secured, the connection's sequence counter is incremented. (Each secured connection has a *sequence counter* that is set to zero when the connection is first secured.)
3. Otherwise, the message is SEC\_PRIVATE or SEC\_AUTHENTICATED. If the connection has not been secured, the connection is secured using the procedure in 4.7.5.3
4. The kernel will make an authenticator using the address of the target host, a timestamp (t), and the incremented sequence number (s+1) --- all of which are encrypted by the session key for the connection:

[DAddr,t,s+1]Sessionkey

5. For SEC\_PRIVATE messages, the kernel will encrypt the data portion of the message.
6. The message is sent down the connection and the sequence counter is incremented.

#### 4.7.5.3 Getting the Kernel to Kernel Ticket

This is the procedure two kernel use to secure a connection between themselves. At the end of the procedure, both kernels will possess a session key known only to them (and the authentication service). The source kernel will also be certain that the target kernel is properly registered with the authentication service, and is not a host unauthorized to run Cronus. The requests sent in the following procedure use SEC\_NONE.

1. The requesting kernel (RK) sends a request to its cluster's authentication service with the address of the destination kernel (DAddr) and a timestamp (t), all encrypted by the secret key of the requesting kernel (RKkey). Each Cronus kernel in multicluster Cronus has a secret key known only to it and its cluster's authentication service.

[DAddr,t]RKkey

2. The authentication manager decrypts the message and generates a session key (skey) and an expiration date for the ticket (exp).

3. If the destination kernel is in the same cluster, the authentication manager will be able to make the ticket for the destination kernel using the address of the requesting kernel (RAddr) and the secret key for the destination kernel (DKkey). The ticket has the following format:

[RAddr,exp,skey]DKkey

If the destination host is in a remote cluster, then the authentication manager obtains the ticket from the remote cluster's authentication service as follows. If the destination host's cluster is not known, a request is first sent to the host asking it what cluster it is in.

- A. The local authentication manager will send the address of the requesting kernel, the address of the destination kernel, the session key, the expiration date, and the timestamp encrypted by the secret key shared by the authentication services in the two clusters (Authkey). If hosts in two clusters are to communicate with each other securely, a mutual secret key must first be established for use between the two authentication services. The authentication manager determines the appropriate Authkey to use based on the ID of the remote cluster.

[DAddr,RAddr,exp,skey,t]Authkey

- B. The remote cluster's authentication manager will decrypt the message and form a ticket for the destination kernel encrypted by the key of the destination kernel, as in Step 3.
- C. The remote cluster authentication manager will increment the timestamp and return the message back to the first authentication manager:

[t+1, [RAddr,exp,skey]DKkey]Authkey

- D. The local authentication manager will then check the timestamp increment and extract the ticket.

The following will then be returned to the requesting kernel:

[DAddr,exp,skey,t+1, [RAddr,exp,skey]DKkey]RKkey

The requesting kernel unwraps the above using its secret key. It extracts the session key (skey), and the ticket. It checks the timestamp and DAddr for evidence of foul play. Then it sends the ticket together with an authenticator to the destination kernel. The destination kernel unwraps the ticket with its secret key, which reveals the session key. It checks the authenticator for evidence of foul play. At this point, both kernels possess the session key and can transmit messages in privacy.

#### 4.7.5.4 Incoming Reliable Message

The procedure used to process incoming reply or request messages is as follows.

1. If the connection was secured, the kernel increments the sequence counter. If security is not used it will hand the message to the destination process with the access group set (AGS) (the identity of the sender) set to NULL.
2. Otherwise, the kernel will unwrap the authenticator using its session key and check it. If the message is a reply, the message is rejected if its security does not match that of its request. In

addition, the authenticator for a reply includes the sequence number on its request. If this does not match the actual sequence number on the request, the reply is rejected.

3. The kernel will decrypt the data part of the message for SEC\_PRIVATE messages.
4. The kernel then hands the message to the proper local process.

#### 4.7.5.5 Encryption Algorithm

We had intended to use the DES encryption algorithm. However, the slow speed of software DES encryption presented several problems. First, performance of the system significantly suffered. *Hardware DES encryption would significantly boost performance.* Second, the slow speed of DES encryption meant the Cronus Kernel became unresponsive for significant periods of time. The encryption of a message is not multiplexed with other activity in the kernel. A local process attempting to communicate with the kernel while it is busy encrypting a large message will not receive a response for a significant amount of time. The local process will eventually time out and incorrectly assume the kernel is dead. The process then returns "ipc failure" errors for RPC's. To work around these problems, the system currently uses exclusive-or encryption. If in the future the kernel is improved so that it multiplexes encryption with other activity, software DES can be used. It is interesting to note that other distributed systems have encountered similar problems and were forced to use exclusive-or encryption as a work around [SAT89].

## Chapter 5. Conclusions

Multicluster Cronus supports organizations wishing to cooperate and share distributed services, but still retain autonomy and control over their own part of the distributed environment. This report discussed why this was not possible in older versions of Cronus. Multicluster Cronus models a set of hosts operating under an autonomous authority as a *cluster*.

Clusters establish service-sharing relationships among themselves by explicitly declaring *imports*, *exports*, and *service domains*. These relationships are stored in each cluster's configuration service.

The basic supporting mechanism for multicluster operation is the Locator. The Locator was totally redesigned for Multicluster Cronus. The Locator makes use of the information stored in the configuration service to narrow the search for an object. The Locator is capable for finding an object regardless of the cluster it resides in.

Applications sometimes need control over what cluster a request is carried out in. To permit this, the Cronus INVOKECONTROL structure was expanded.

Older versions of Cronus suffered from a limit on the maximum number of concurrent reliable connections a kernel can have open. This limit was overcome in Multicluster Cronus. Overcoming this limit makes application development much easier since the number of reliable connections needed is now irrelevant. Previously, an application developer would have to use unreliable, instead of reliable, communication when there was a danger of hitting the limit.

Authentication was also improved in Multicluster Cronus. Authentication suggests several avenues for future study and work. First the kernel should be upgraded so that it multiplexes encryption activity with other activities. Second, the scheme could be extended to include unreliable, as well as reliable, messages. Currently unreliable messages are all unauthenticated and privacy is not supported. Third, the scheme could also be extended to include direct connections. The protocol used to establish a direct connection is authenticated, but authentication of each message and privacy on a direct connection are not supported. By supporting private direct connections, much encryption activity could be offloaded from the kernel, by having the system prefer to send private messages over direct connections.

## Chapter 6. References

[BS89] Berets, J., Sands, R., "Introduction to Cronus," Technical Report 6986, BBN systems and Technologies, January 1989.

[SAT89] Satyanarayan, M., "Integrating Security in a Large Distributed System," ACM Transactions on Computer Systems, Vol. 7 No. 3, Association for Computing Machinery, August 1989, page 272.

[WFN90] Walker, E., Floyd, R., Neves, P., "Asynchronous Remote Operation Execution in Distributed Systems," Proceedings of the 10th Int'l Conference on Distributed Computing Systems, IEEE Computer Society, May 1990, pp. 253-259.



**MISSION  
OF  
ROME LABORATORY**

Rome Laboratory plans and executes an interdisciplinary program in research, development, test, and technology transition in support of Air Force Command, Control, Communications and Intelligence (C<sup>3</sup>I) activities for all Air Force platforms. It also executes selected acquisition programs in several areas of expertise. Technical and engineering support within areas of competence is provided to ESD Program Offices (POs) and other ESD elements to perform effective acquisition of C<sup>3</sup>I systems. In addition, Rome Laboratory's technology supports other AFSC Product Divisions, the Air Force user community, and other DOD and non-DOD agencies. Rome Laboratory maintains technical competence and research programs in areas including, but not limited to, communications, command and control, battle management, intelligence information processing, computational sciences and software producibility, wide area surveillance/sensors, signal processing, solid state sciences, photonics, electromagnetic technology, superconductivity, and electronic reliability/maintainability and testability.